CS-320 Project Two Journal

Mike R. Montminy

Southern New Hampshire University

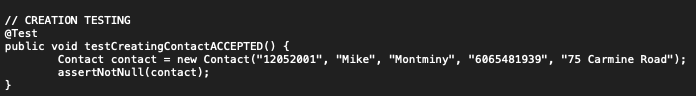
CS-320: Software Testing, Automation, & QA

Dr. Angel Cross

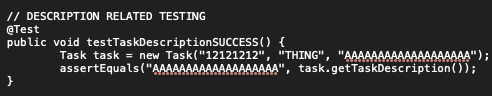
06/22/2025

**CS-320 Project Two Journal**

My JUnit testing approach so far has lined up with the software requirements well in a few ways. All tests that I have made include making sure that variables within the specific parameters like the length requirements. For example, the contact adding needed to make sure parameters such as first name, phone number, and address of that contact were within the requirements for overall string length. To help with that, I set up a ‘CreatingContactACCEPTED’ test where it will ensure that all the parameters are at or below the required lengths.

 In terms of the overall quality of my JUnit tests, I would say they are solid and are able to meet the expected requirements. My overall goal was to make sure that these tests covered as many public methods as possible. Each test goes through either a failure or success or setting variable, getting variables, adding tasks/contact, updating them, etc. In terms of the actual percentage of coverage, the Task and TaskService testing files were able to trigger most of the possible execution paths and make sure everything is working smooth.

I made sure that my testing code was technically sound in a few different ways. First, I made certain that all valid inputs would be confirmed with an assertEquals function to confirm the getter for the taskID would return the correct value. In my TaskDescriptionSUCCESS test in the TaskTesting file I have it set up to create a new task with the given description being less than 50 characters, and then having it use assertEquals to confirm that value is grabbed by the getter properly.



Making the code efficient was a bit hard to do with the assignment, especially with the testing files. For the task service items like adding, updating, and deleting the tasks I chose to use a hash map as it was in my opinion the easiest to implement for this type of data needing to be stored somehow. To make any of the CRUD style tasks efficient, my goal was to have as few lines of code as possible while ensuring proper functionality. I have it set up that if there is an attempt to add a task with the same ID as another, the error is immediately thrown, and then if there isn’t a matching ID, the task gets immediately put into the data structure.

 For these three milestones I had to make a few decisions of what other types of testing I wanted to perform to ensure maximum testing coverage percentage. Unit testing specifically centers around testing specific chunks, or in this case units, of code with certain parameters to ensure functionality. These were used all throughout the testing files for the three projects of tasks, contacts, and appointment applications. The main characteristics of a JUnit testing setup are that they are heavily isolated, exponentially faster than most other test types to run, and can be automated with little to no difficulty. The second type I chose to implement was boundary testing in cases where there were character or digit limits where they couldn’t exceed a certain amount. Boundary tests help to check for items such as validation logic issues and random one-off scenarios. A third form of testing that I used in these milestones was negation or negative testing by giving the test inputs null values to ensure errors are thrown when a user enters a null value. These tests help to confirm that error handling and validation logic is how it should be.

Some test types weren’t applicable due to project constraints. For instance, system testing wasn’t applicable because there was no UI to simulate a full workflow. Additionally, integration testing wasn’t relevant since the milestone classes functioned independently without any real inter-class communication. Each testing type has clear use cases: unit testing supports early bug detection and easier refactoring in OOP-based systems; boundary testing is essential for input-heavy applications to improve robustness; and negative testing ensures strong validation logic. Although not used, integration testing is key for verifying data flow in complex systems, while system testing is ideal for validating end-to-end behavior in user-facing apps.

I had to approach this project being heavily detail oriented for designing the testing cases regarding input validation and error handling. For example, confirming that the ContactService was properly rejecting any IDs longer than the set ten-character limit instead of using assumptions. Understanding the interrelationships between the classes was essential in testing the TaskService file and ensuring that all IDs, descriptions, and names were handled separately yet still worked in a single object. Regarding reducing bias, I made sure to check each method behavior within the AppointmentService rather than relying on memory and having to fight several issues down the line. This was especially useful when I had to draft test cases for null inputs. Having to maintain discipline was a key part of ensuring all test cases were met. Instead of skipping negative tests, I chose ot write them to ensure all invalid cases were covered for all items. This helps to reduce the chance of unexpected errors, and the overall development debt has a chance to be drastically reduced.

**References**

Montminy, M. R. (2025). Screenshots of input validation methods in Java [Unpublished screenshot]. Personal collection.